Module 1: Introduction

Exercise

1. Login to Jupyter: https://jupyter.math.uwaterloo.ca/
2. Create a new Notebook for “Python 3 (ipykernel)”.
3. Copy the following command from Learn, then paste it into Jupyter:

```
!wget --backups=5 https://student.cs.uwaterloo.ca/~cs114/src/module-01-start.ipynb
```

Welcome to CS 114

Most information about the course is in Learn.

The course outline is at https://outline.uwaterloo.ca/view/nnw88w

Contact info for all course staff is there, but if in doubt, here are the key email addresses:

Instructor	Cameron Morland – cjmorland@uwaterloo.ca
Coordinator	Scott Freeman King – sfking@uwaterloo.ca
ISAs and IA	– cs114@uwaterloo.ca

In this course we will be using Python inside Jupyter.

Course components

Class slides are available on the course web site:
https://student.cs.uwaterloo.ca/~cs114/slides/

The textbook is available in UW Online. It includes extra explanation and extra exercises.

Quizzes are to be completed in the textbook. They are due most weeks, on **Wednesdays**.

Assignments are available on the course web site. These are due most weeks, on **Fridays**.

Exams The midterm and final exams will be in-person, and involve both code and theory. More details as we get closer.

Marking Scheme

Details are in the outline: https://outline.uwaterloo.ca/view/nnw88w

Review the Policies in the outline. In particular:

> *"In order to maintain a culture of academic integrity, members of the University of Waterloo community are expected to promote honesty, trust, fairness, respect and responsibility. [See www.uwaterloo.ca/academicintegrity for more information.]"*

Assignments

Most of your learning comes from struggling with material. You learn little from merely copying work, or even ideas, from another.

! All assignments are to be done individually.

! Don't use ChatGPT or other systems to help with your assignments.

!

You must do your own work in this course.

- Don't look at someone else's programs written for an assignment.
- Don't show your programs to anyone except course staff.
- Don't discuss our assignments with people outside of the course. Ask course staff instead! We're available in discussion, office hours, and by email.
- If you discuss assignments with others, do not keep any written material.

Assignments

A few pieces of advice:

- Start your assignments early.
- Make sure you have time to test thoroughly, and fix your code when needed! Bring questions to the office hours or discussion board as soon as possible.
- Our submission system, MarkUs, is set up to identify certain errors with your code.

> **!** MarkUs will reject any code in which it can identify errors.
> Submit early enough that you can fix the errors it finds.

- Go over your graded assignments: learn from your mistakes.

A Subset of Python

Python is a big, complicated language.

We want to become skilful with specific parts of it, and we want to fully understand the parts we study. Most of you have not used Python before. Use the tools that we learn!

If you have used Python before, you may have used tools not in this course.

We want to learn the tools in the course; we want to become proficient with those tools!

> **!** Use only features presented in the class slides.
> Solve problems using the techniques used in the course.
> (For example, do not use `break`.)

MarkUs will give you a message and reject any code that uses techniques outside the course.

What is Computation?

A computer program is a set of instructions to complete a particular task.

Many tasks are mathematical: the computation of certain mathematical values.

Many mathematical questions we can answer by hand. For example:

Exercise

How many natural numbers is 12 divisible by?

Carefully consider: how did you solve this?

A big part of our task as programmers is figuring out how to take a problem like this, and turn it into an **algorithm**: an explanation of how to do it.

The specific instructions to the computer are not the hard part; the hard part is figuring out how to explain the task at all.

What is Computation?

By hand, the answer is six: $\{1, 2, 3, 4, 6, 12\}$ are the only natural numbers that divide 12.

But if I were to ask the same question of another number, I might not be able to do it by hand.

> **!** How many natural numbers is 5 218 303 divisible by?

By hand, this question is too difficult to solve in a reasonable amount of time. Our task in this course will be to write instructions to allow the computer to solve tasks such as these.

Instead of solving this particular problem, we will write a **function** that solves the problem in general, for any number.

We can **test** our function using small numbers like 12, 31, and 63. Once we are confident it is correct, we can use it to answer the "big" question.

Exercise

Continue working in your group. Immediately go to:

https://jupyter.math.uwaterloo.ca/

1. Log in.
2. In the Launcher, click "Python 3"; this will create a new file called "Untitled.ipynb".
3. Once this is working, run the following incantation s(copy & paste from Learn):
   `!wget --backups=5 https://student.cs.uwaterloo.ca/~cs114/src/module-01-start.ipynb`
4. Close your `Untitled.ipynb` tab.
5. Double click on the file `module-01.ipynb` to edit it. Work on class exercises in this file.

Simple calculations in Jupyter

Our basic mathematical operators are: `+`, `-`, `*` for multiplication, `/`, and `**` for exponentiation. We can add round brackets `()` as needed.

(Other brackets including `[]` and `{}` have special meanings. We'll see them later.)

When we select a cell and hit shift-enter, or click the ▶ symbol, it runs the code, and displays the last calculated value.

Exercise

In Jupyter, calculate the value $\frac{4(7+3)^2}{1+1}$

Printing more than one value

Jupyter shows only the last value. If we want to see more than one value, we use the `print` function:

```
print(6 * 7)
print(42 / 7)
```

This will display two values, each on a separate line.

Get in the habit of using `print` to inspect values.

We can also print more than one value on one line, giving more than argument to the `print` function:

```
print(6, 7, 6*7)
```

This will print `6 7 42`, all on one line.

Introducing strings

A **string** is a way of storing a collection of characters. This could be a word, or several words, or a paragraph.

In Python there are several ways to create a string, but the simplest is just to use quotation marks:

```
print("hello world!")
print("A force of 1.4 N applied over 2.72 m does", 1.4 * 2.74, "J of work.")
```

There are many ways we can use strings, but for now, we will use them to help make our code output more understandable.

Exercise

Use `print` statements to display a message describing the kinetic energy of a 4.2 kg mass moving at 2.4 m/s. (Recall: $E_k = \frac{1}{2}mv^2$)

A simple Physics problem

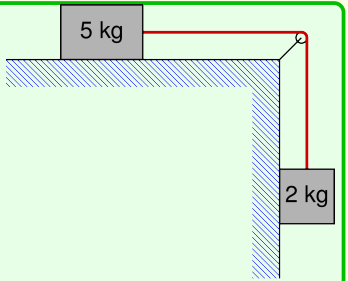
Exercise

A 5 kg goose is standing on ice on the top of Dana Porter library, connected using rope and a pulley to a 2 kg duck which is sliding down the wall.

Neglecting air resistance and friction:

1. How fast are the birds accelerating?
2. What is the tension in the rope?

Assume the gravitational pull on campus is 9.8 m/s^2.



Variables

To solve any significant problem, we need to be able to do a **calculation**, **store** the result, and then use the result to figure out what to do next.

In Python we can **assign a value** to a **variable** using the "=" operator.

You should pronounce "=" as "gets".
Using this to help solve our problem:

```
Mduck = 2.0
Mgoose = 5.0
Fg = 9.8 * Mduck
print("Fg is:", Fg, "N")
mtotal = Mduck + Mgoose
accel = Fg / mtotal
print("accel is:", accel, "m/s**2")
```

Exercise Extend this code to solve the goose/duck problem so that at the end a variable `Ft` stores the tension in the rope.

Variables

Note that in Python the `=` symbol means something quite different from $=$ in Math.

In Math, we can say things like $x + 5 = 4x - 1$. But in Python we can only have the **name of a variable** on the left side of `=`.

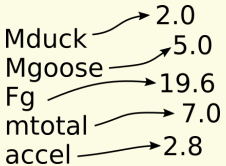
To pronounce a statement like `x = 3 + 4`, we say "x gets 3 plus 4", or "x gets 7".

Step by Step evaluation and State Diagrams

To interpret a snippet of code, the computer walks through it, line by line, and does each step. It's sort of like reading a recipe.

```
Mduck = 2.0
Mgoose = 5.0
Fg = 9.8 * Mduck
print("Fg is:", Fg, "N")
mtotal = Mduck + Mgoose
accel = Fg / mtotal
print("accel is:", accel, "m/s**2")
```

As the computer steps through the code, it keeps track of the value of each variable, in a **table**, that we can visualize as a **state diagram**.

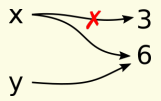


Step by Step evaluation and State Diagrams

If we assign a new value to a variable, it now refers to a different value:

```
x = 3
print("x is:", x)
x = x + 3
print("x is:", x)
y = x
print("x is", x, "and y is", y)
```

The state diagram changes.
Notice: we do not overwrite the values. We **remove the arrow, and make a new one**.



!

This will matter later, so get in the habit of thinking this way.
Using = changes the arrows, not the values.
If we assign a variable to another variable, they are both arrows pointing at the same value.

Step by Step evaluation and State Diagrams

Exercise

Use a state diagram to track the values of all the variables:

```
foo = 3 * 2
bar = foo
foo = bar * 2
qux = foo + bar
```

Mathematical functions

So far, we can do some calculations, but only using the basic operations (`+`, `-`, `*`, `/`, etc), like a simple calculator.

To turn our Python into a "scientific" calculator, we will add one line at the top of our code:

```
import math
```

This gives us access to many mathematical functions, including `math.sin`, `math.cos`, `math.asin`, `math.log`, and more:

```
print("The cosine of 1 (radians) is:", math.cos(1.0))
```

Exercise Use Python to calculate the value of $\dfrac{\sin(1+\cos(2))}{\tan(3)}$

The function `math.radians` converts an angle in degrees to radians.

`print(math.sin(math.radians(45)))` displays `0.707...`

Read The Fine Manual

The `help` command will show us the documentation:

- `help(math)` displays help on the `math` module
- `help(math.cos)` displays help on the `math.cos` function

Ex. Use the documentation of the `math` module to see how to use Python to calculate $\log_3 30$.

With the `math` module, we can do anything we can do with a scientific calculator. But if we want to do similar calculations, we need to duplicate our work.

Suppose we want to calculate the tension twice, with a 2 kg duck and a 3 kg duck:

```
Mduck = 2.0
Mgoose = 5.0
Fg = 9.8 * Mduck
print("Fg is:", Fg, "N")
mtotal = Mduck + Mgoose
accel = Fg / mtotal
print("accel is:", accel, "m/s**2")
```

```
Mduck = 3.0
Mgoose = 5.0
Fg = 9.8 * Mduck
print("Fg is:", Fg, "N")
mtotal = Mduck + Mgoose
accel = Fg / mtotal
print("accel is:", accel, "m/s**2")
```

One important aspect of computer science: **be lazy**.
If we're working, we're doing something wrong.

Writing something twice is **work**.
There should be a better way, and there is: we can define a new **function**.

Defining Functions

An example function definition:

```
def duck_goose ( Mduck , Mgoose ) :
    Fg = 9.8 * Mduck
    print("Fg is:", Fg, "N")
    mtotal = Mduck + Mgoose
    accel = Fg / mtotal
    print("accel is:", accel, "m/s**2")
```

It contains:

1. the keyword `def`,
2. the name of our new function,
3. inside brackets `(` , `)`: zero or more parameters,
4. a colon `:`,
5. an indented block of code.

The definition itself does nothing.

We need to **call** the function, like:

```
duck_goose(2.0, 5.0)
```

or

```
duck_goose(2 + 1.7/2, 2.0**3)
```

A **function call** contains:

1. the name of the function,
2. inside brackets "`()`", **arguments** corresponding to the **parameters**.

This way we can call it with different values for `Mduck` and `Mgoose`.

Exercise

Modify the following `duck_goose` function so the gravitational pull `g` is also a parameter, instead of always being 9.8 m/s^2.

```
def duck_goose(Mduck, Mgoose):
    Fg = 9.8 * Mduck
    print("Fg is:", Fg, "N")
    mtotal = Mduck + Mgoose
    accel = Fg / mtotal
    print("accel is:", accel, "m/s**2")
```

Exercise

Imagine a 2 kg duck is wearing a 1.8 kg space suit, and a 5 kg goose is wearing a 4 kg space suit. **Don't change your function.** Call it twice to determine:

1. The acceleration of the birds on Mars, where the gravitational pull is 3.721 m/s^2;
2. The acceleration of the birds on the Moon, where the gravitational pull is $\frac{1}{6}$ of what it is on Earth.

Functions calling functions

A function can call any function that is defined, even one we wrote.

Now that our `duck_goose` function has three parameters, we can write another function to do the calculation more than once:

```
def duck_goose_twice(mass1, mass2, pull1, pull2):
    print("On a planet where the gravitational pull is", pull1)
    duck_goose(mass1, mass2, pull1)
    print("On a planet where the gravitational pull is", pull2)
    duck_goose(mass1, mass2, pull2)
```

Defining Functions

The momentum of an object of mass m and velocity v is given by $p = mv$.

The kinetic energy of this object is given by $E_k = \frac{1}{2}mv^2$.

Exercise

Write a function `basic_properties(m,v)` that displays two messages, indicating the momentum and kinetic energy of an object.

For example, `basic_properties(3,5)` should print something like:

```
The momentum is 15 kg*m/s
The kinetic energy is 37.5 J
```

Fruitful functions

Consider this code:

```
v = 4 + math.cos(0)
print(v)
```

The math functions, such as `math.cos` and `math.log`, don't print anything on the screen. Instead, they **return** a value, and the code that **called** can then do something with the value: print it, or do more arithmetic, or whatever.

To do this, we use the `return` statement. While reading a program "like a recipe", the moment we encounter a `return` statement, we stop evaluating the program. Imagine the "call" has that value.

To write a function like $d(x) = 2x$:

```
def double(x):
    return 2 * x

y = double(18) + 6
print(y)
```

Exercise

Write a function `area` that takes the radius of a circle, and returns its area.
Recall that the area of a circle is given by $A(r) = \pi r^2$.
Use the constant `math.pi`.

Use your function to calculate the total area of a collection of circles: what do you get when you call

```
print(area(1) + area(2) + area(3))
```
?

Incremental Development

We want a function to calculate the distance between two points (x_1, y_1) and (x_2, y_2).

1. Start by making a function that does not work, but returns something.
```
def distance(x1, y1, x2, y2):
    return 0.0
```
2. Check that it does what we expect.
```
print(distance(0, 0, 4, 3))
```
3. Add code to calculate a partial answer; debug to make sure the pieces make sense:
```
def distance(x1, y1, x2, y2):
    xdiff = x2 - x1
    ydiff = y2 - y1
    print("xdiff is", xdiff, "and ydiff is", ydiff)
    return 0.0
print(distance(10, 10, 14, 13))
```
4. Make small changes and repeat. **Make it sure makes sense to you.**
5. Once debugged, remove extra `print` statements.

The Docstring

Recall that if we say something like `help(math.cos)`, it shows us information on how to use the `math.cos` function. But if I ask for `help(distance)`, it displays very little information.

To fix this, when we write a function, we start by writing a **docstring** that explains **what the function is supposed to do**. As the **first statement** in a function, we put a string that contains the help message. Write this string using triple-double quotes `"""like this"""`.

```
def distance(x1, y1, x2, y2):
    """Calculate the  distance between (x1, y1) and (x2, y2)."""
    xdiff = x2 - x1
    ydiff = y2 - y1
    return math.sqrt(xdiff**2 + ydiff**2)
```

! You must write a docstring for every function you write.
We **will not mark** code that is not properly documented.

Docstring conventions

The Python community has developed conventions that make docstrings easier to understand.

- **Always** refer to each parameter **by name**, and say what the parameter is for.
  Don't write something like `"""Calculate the distance between the two points."""`.
- Write your sentence as if you are **telling** the function what to do.

  Write something like: `"""Return the area of a circle of radius r."""`.

  Don't write: `"""Returns the area of a circle of radius r."""`.
- Don't describe what it does or how it does it.
  Don't write something like `"""Return pi * r**2."""`.

Remember, this string must be the **first statement** in the function.
No code may come before it.

When you call `help` on your function, it will display your docstring.

Docstring conventions

An example:

```
def refraction_angle(theta1, c1, c2):
    """Following Snell's law, calculate the angle that a ray of light at
    angle theta1 takes, when it moves from a medium with speed of
    light c1 to a medium with speed c2.
    """
    return math.asin((c2 / c1) * math.sin(theta1))
```

The community describes this, with more detail, in PEP 257 - Docstring Conventions.

Type Annotations

So far, the only values we have seen are numbers, like `42` and `3.14`, and strings such as `"hello world"` and `"""Return the area of a circle of radius r."""`

In Math class, we often make a distinction between **integers** ($\mathbb{Z}$) like -3, 0, and 42, and **real numbers** ($\mathbb{R}$) like 3.14, π, and $\sqrt{2}$.

In Python, these values are of different **types**. When we write functions, we want to indicate what type its parameters are, and what type the value is that it returns.

We use the type `str` for strings, `int` for integers, and `float` for real numbers.

Type annotation syntax

The type annotations are part of the `def` statement. Example:

```
def distance(x1: float, y1: float, x2: float, y2: float) -> float:
    """Calculate the  distance between (x1, y1) and (x2, y2)."""
    xdiff = x2 - x1
    ydiff = y2 - y1
    return math.sqrt(xdiff**2 + ydiff**2)
```

1. To indicate the type that each parameter takes, after the name of each parameter, write a colon and a space (`: `) then the type.
2. To indicate the type that the function returns, between the close bracket and the colon, write a little arrow surrounded by spaces (`->`) followed by the type.

Exercise

Annotate the function `area`:

```
def area(r):
    """Return the area of a circle of radius r."""
    return math.pi * r**2
```

What is the difference between `int` and `float`?

Consider how to annotate these:

```
def force(mass, accel):
    """Return the force needed to accelerate mass kg at accel m/s**2."""
    return mass * accel
def total_electrons(n, peratom):
    """Count electrons in n atoms, each containing peratom electrons."""
    return n * peratom
```

`mass` and `accel` are continuous values; we can change them by any amount.

`n` and `peratom` are discrete, whole values; we can only change them by integer amounts.

So we write:

```
def force(mass: float, accel: float) -> float:
def total_electrons(n: int, peratom: int) -> int:
```

If it's a measurement, it's `float` (e.g. 3.2 N, 1.107 kg/l, or 6.0 C/s)

If you're counting something, it's `int` (e.g. 32 He atoms, 12 monkeys, or a charge of −1 e.)

Testing exact values with `check.expect`

In creating `distance`, we used the idea of **incremental development**.

This idea is so important that programmers have created tools to make it easier to do.

When we ran the first cell of the module 01 start code, it installed the `check` module.

To use it, at the top of your code write:

```
import check
```

Now we have a new function called `check.expect`. It takes three arguments:

1. a `str` that we use as a name for the test,
2. an expression that is a call to the function we are testing,
3. an expression that is the value the function call should return.

So we might say:

```
check.expect("a few atoms", total_electrons(5, 10), 50)
```

Testing inexact values with `check.within`

We work often with measurements, and these are **`float`** values.

But **`float`** arithmetic is slightly inexact; it is not the case that `0.2 + 0.1` equals `0.3`.

Any time we work with **`float`** values, we will use `check.within`. It is the same, but takes a fourth argument. It takes:

1. a **`str`** that we use as a name for the test,
2. an expression that is a call to the function we are testing,
3. an expression that is the value the function call should return.
4. a positive **`float`** value that says **how close** the two values must be to pass the test.

So we might say:

```
check.within("a little force", force(5.0, 10.0), 50.0, 0.0001)
```

The test passes if `force(5.0, 10.0)` is between `50.0 - 0.0001` and `50.0 + 0.0001`.

Write some tests

Exercise

Fill in the blanks in the start code to write two tests for the provided `distance` function.

!

You must use the `check` module to test every function you write!
We may provide a few tests to show how a function should behave. For every function you write, you must create at least two tests, **in addition to any we provide.**

For most functions, 2 tests will not be enough to thoroughly test it. You should test carefully to ensure you code always works correctly. Code that does not work will usually get a grade of 0.

Make tests match annotations

Your tests should always match the annotations.

It is incorrect to write something like:

```
check.within("a few atoms", total_electrons(5.0, 10.0), 50.0, 0.0001)
```

or

```
check.expect("a little force", force(5, 10), 50)
```

Watch especially with zero: `0` is an `int`, while `0.0` is a `float`.
You cannot use `0` and `0.0` interchangeably!

The following test is incorrect, and if your code passes it, your code is incorrect.

```
check.expect("a little force", force(5.0, 0), 0)
```

The Design Process

When writing a function, using this process will save effort.

1. Write the **docstring** and **annotations**.
   If you can't, you don't what the function is supposed to do; figure that out first.
2. Write **tests**, using `check.expect` or `check.within`.
   Calculate the values by hand, to help ensure you know what you are trying to do.
3. Only after completing these steps, start writing code.

As you get into the problem:

1. consider if you are actually solving the problem. If not, you may want to change your docstring or tests.
2. look for a **smaller problem** that is a part of the whole problem. You may want to write a separate "helper" function that solves this smaller problem.

Exercise

If a, b, and c are the lengths of the three sides of triangle, then the area of that triangle is given by **Heron's Formula**,

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = \frac{a+b+c}{2}$.
Write a function `area(x1, y1, x2, y2, x3, y3)` that takes the coordinates of the three corners of a triangle, and returns its area.

Hint

Start by writing the **docstring** and **type annotations**.
Then write some tests. **Choose some triangles that are easy to work with.**
Finally, use **incremental development** to finish your solution.

Module summary

- You should be able to define and use variables and simple arithmetic functions.
- Start getting used to error messages.
- Use state diagrams to help work out what a piece of code does.
- Use docstrings, annotations, and tests to make your code easier to create and understand.

Before we begin the next module:

- Read and complete the exercises in module 1 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 1 Review Quiz, due soon.